A DESCRIPTION OF SEVERAL TOOLS FOR THE SYNCHRONIZATION OF CONCURRENT PROCESSES

by

Brian A. Barsky and Christian Gram

UUCS-80-102

March 1980

A DESCRIPTION OF SEVERAL TOOLS

FOR THE SYNCHRONIZATION OF CONCURRENT PROCESSES

by

Brian A. Barsky and Christian Gram*

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112
U.S.A.

*On sabbatical leave from:
Department of Computer Science
Technical University of Denmark
DK-2800 Lyngby, Denmark

Abstract

Concurrent processes are tasks which may be executed simultaneously. When several such processes have access to shared variables, it is necessary to establish some regimen to control this access. Several language tools for expressing various synchronization disciplines are presented.

Table of Contents

0. Introduction

Concurrent processes are tasks which may be executed simultaneously. This is distinct from sequential processes which are executed sequentially; that is, one process cannot start until the preceding one has terminated. Statements contained in a cobegin/coend block define concurrent processes.

Concurrent processes which do not have any variables in common and do not communicate in any way are called independent or noninteracting. If a variable is altered by one of the concurrent processes, then the other processes are not permitted to access it.

More generally, concurrent processes are interacting; that is, they have access to common variables representing shared resources. In order to share resources in an orderly way, it is necessary to synchronize concurrent processes with respect to the use of the shared resources. A variable common to (i.e., used by) several concurrent processes is called a shared variable.

The following sections briefly describe some language tools that make it possible to express some types of shared (but controlled) access to common variables from concurrent processes.

1. Critical Regions

It is desired to guarantee that only one process operate on any given common variable at any given time. The principle of mutual exclusion states that a statement involving a shared variable can only be executed if no other statement involving the same shared variable is currently being executed.

A critical region associates a statement with a shared variable. If several critical regions are associated with the same shared variable, then at most one of them can be executing at any given time. This does not, however, preclude several critical regions executing simultaneously if each is associated with a distinct shared variable.

A language construct that declares that a variable v is of type T and is to be shared among concurrent processes is

var v: shared T;

The shared variable v is accessible only inside critical regions which have been associated with it. Let P be one of the concurrent processes. Then the statement S in P can be defined as such a critical region by the construct

region v do S;

which has the following semantics. If no other critical region refferring to this variable v is currently executing, the statement S will be executed. Otherwise, the process P will be delayed until v becomes "free", at which time S will be executed.

Note that deadlock can occur when using nested critical regions. Processes are deadlocked when each one is waiting indefinitely for an event that will never happen. Consider, for example, the following program segment:

```
var v:  shared V;
    w:  shared W;
cobegin
/* P */ region v do region w do ...;
/* Q */ region w do region v do ...
coend;
```

If both processes P and Q enter their regions v and w, respectively, at the same time, then neither process can enter its next region. Specifically P cannot enter its region w because Q is inside its region w, and Q cannot enter its region v because P is inside its region v. Hence each process will wait indefinitely for a resource possessed by the other one.

However, if critical regions are consistently nested such that the shared variables are reserved in a specific order (common to all the concurrent processes), it can be shown that deadlock cannot occur.

2. Message Buffers

The processes whose interactions are controlled by critical regions can ignore the other processes except that they must have exclusive use of a shared resource. Processes cooperating on common tasks require more direct interactions; that is, they must be able to communicate information between them.

This information is often exchanged in discrete data entities called messages. A producer process sends them to a consumer process which receives them. Suppose the producer sends a message prior to when the consumer is capable of receiving it. Rather than forcing the producer to be delayed until the consumer is ready to receive it, it is preferable to enable the producer and consumer to proceed at rates independent of each other. This is accomplished by introducing a temporary storage area, called a buffer, which stores the messages sent by the producer until the consumer is ready to receive them. This arrangement is subject to two constraints: first, the producer cannot exceed the finite capacity of the buffer; and second, the consumer cannot receive messages faster than they are sent. This is accomplished as follows: If the buffer is full, the producer is required to wait until the consumer removes a message from the buffer. If the buffer is empty, the consumer must wait until the producer puts a message into the buffer.

A language construct for handling this type of problem is a special buffer data type and two procedures "send" and "receive" for operating on buffers. As an example, after declaring the variables

```
var B: buffer max of T;
var MESS1, MESS2: T;
```

Concurrent processes may issue procedure calls such as

send (MESS1, B) and receive (MESS2, B).

Mutual exclusion is guaranteed because only one procedure call at a time can be executed on each buffer (the buffer acts as a shared variable), and send and receive will also delay the calling process if the buffer is full or empty, respectively.

3. Semaphores

Oftentimes, the only communication necessary between processes is the sending and receiving of a timing signal. A timing signal can be thought of as a special case of a message where the content is irrelevant; all that matters is that it has been sent and, perhaps, how many times this occurred. Hence it suffices to replace the buffer with a nonnegative, integer-valued variable which counts the number of signals sent, but not yet received.

The synchronization of processes requires the capability of having one process wait for a signal sent by another process. This is accomplished by introducing variables of a new type semaphore. A semaphore v can be conceptualized as consisting of four components: three nonnegative, integer-valued variables initialized to zero and a queue of processes. The number of initial signals will be represented by c(v), s(v) is the number of signals sent, and r(v) is the number of signals received. (Note that in the usual implementation of semaphores these three quantities are combined into a single integer variable.) Since the number of signals received cannot be more than the number of initial signals plus the number of signals sent,

$$r(v) \leq c(v)+s(v);$$

and since the count of the number of initial signals plus the number of signals sent, but not yet received, must not exceed the maximum integer,

$$c(v)+s(v)-r(v) \leq \text{maxint},$$

where maxint is the largest possible integer which can be represented in the particular computer. Combining this information into a single expression, and recalling that c(v), s(v), and r(v) have nonnegative values, any semaphore must satisfy the invariant condition:

$$0 \leq r(v) \leq c(v)+s(v) \leq r(v)+maxint.$$

A semaphore variable v with an initial value i is declared as

var v: semaphore (i);

The primitive operations are the two procedure calls

signal (v) and wait (v),

and v can only be accessed through these operations. These operations exclude each other in time and have the following effects when invoked.

The operation signal (v) executed by process P increments s(v) by 1. Then, if the queue of processes which is associated with the semaphore v is nonempty, one waiting process is selected (according to some scheduling algorithm) and enabled to continue, and r(v) is incremented by 1. The execution of process P is unaffected.

The execution of the operation wait (v) by process Q compares the value of r(v) to that of c(v)+s(v). If r(v) is strictly less, then r(v) is incremented by 1, and process Q continues; otherwise (r(v) = c(v)+s(v)), process Q is suspended and placed on the queue of processes which is associated with the semaphore v.

For example, the following algorithm uses semaphores to synchronize two processes, P and Q, such that the number of times the process P has been begun does not exceed n times that of process Q.

```
var v:  semaphore (0);
    n:  integer;

n := ...;

cobegin

/* P */ while true do
        begin
              wait (v);
              ...
        end;

/* Q */ while true do
        begin
              for i := 1 to n do signal (v);
              ...
        end

end;
```

4. Conditional Critical Regions

A conditional critical region provides a mechanism which has the capability of delaying a process until some arbitrary condition is satisfied by a shared variable. The primitive operation which delays a process until a shared variable v satisfies a condition B(v) of type Boolean is

await B(v).

An await statement must be enclosed by a critical region which is associated with the shared variable involved in the Boolean condition. This construct is thus used in the following way:

```
var v: shared T;
region v do
/* P */ begin
                ...
                await B(v);
                ...
        end;
```

The process P enters its critical region, executes the first part of the critical region, and continues in the usual manner provided that the shared variable v satisfies the condition B(v). If the condition B(v) is not satisfied, the process is suspended and placed in a queue of waiting processes associated with the shared variable v. Note that even though this process is in its critical region, other processes can now enter their critical regions, since P is in the queue. When a process completes its critical region it may have altered the shared variable v, and hence it is possible that the wait condition of one or more

processes in the queue has become satisfied. Thus, whenever a critical region is completed, every process in the queue must be allowed to check if its condition is now satisfied by the shared variable. In this case, one process, whose condition has now become satisfied by the new value of the shared variable v, is allowed to continue, while all others remain in the queue.

5. Event Queues

Resource scheduling requires the ability to insert processes in a waiting queue and remove them again according to some scheduling strategy. As a minimum, this requires the basic tool of queues:

A queue q of elements of type T is declared as

var q: queue of T;

An element t of type T is entered and removed, respectively, from the queue q by the procedure calls

enter (t,q) and remove (t,q).

The entering and removing of elements is performed according to some scheduling policy (that is, not necessarily First In / First Out). Whether or not a queue q is empty can be determined by the truth value of the Boolean function

empty (q).

Explicit control of resource scheduling also requires the capability of transfering processes among the several event queues of processes for each shared variable. Therefore event queues explicitly associated with specific shared variables are introduced. Each event queue should be associated with a distinct event which is being awaited. This unifies the treatment of all processes waiting for the shared variable to satisfy a particular wait condition. Thus an event queue consists of processes which are all waiting for the occurrence of some particular event. An event queue e associated with a shared variable v

is declared as

```
var v: shared T;
    e: event v;
```

A process P can transfered to the event queue e by the procedure call

```
await (e).
```

The occurrence of the event e is signified by the procedure call

```
cause (e).
```

Both procedures may be called only inside critical regions associated with v. This enables all the processes in the event queue e to reenter their critical regions associated with v, and thus one of them will be selected to execute its critical region.

6. Monitors

Monitors provide a language construct which integrates the definition of a common variable with a set of procedures which can operate on it. The only access to the common variable(s) is to call one of the procedures declared in the monitor. The mutual exclusion is obtained by the fact that only one such procedure call at a time will be executed; that is, these operations exclude each other in time. Thus the shared variables are completely invisible to the concurrent processes; these can only call for operations to be performed on the variables through the procedures. A monitor also contains an initial statement which initializes the shared variable.

A monitor can be specified by the language construct

```
monitor
     /* variable declarations */
     /* procedure declarations */
     begin
          /* initial statement */
     end;
```

and the compiler can check that a shared variable is accessed only by the procedures defined in the monitor.

7. Conclusion

Each of the aforementioned synchronizing tools is more appropriate in certain situations than in others. Critical regions guarantee that no more than one process operates on any given variable at any given time. Message buffers are ideal for the communication of information between processes. Semaphores provide a direct mechanism for awaiting a timing signal. The conditional critical region is a natural means of delaying a process until some arbitrary condition is satisfied. Event queues are general synchronizing tools that give the programer explicit control of resource scheduling, but at the expense of necessitating concern for details of scheduling. The monitor is an elegant language construct which protects the accessibility of one or several shared variables by uniting them with a set of procedures which can operate on the variables.

REFERENCES

1. Brinch Hansen, Per. Operating Systems Principles, Prentice-Hall, Inc., Series in Automatic Computation, Englewood Cliffs, N.J., 1973.

2. Brinch Hansen, Per. The Architecture of Concurrent Programs, Prentice-Hall, Inc., Series in Automatic Computation, Englewood Cliffs, N.J., 1977.

3. Habermann, A. Nico. "Synchronization of Communicating Processes", Communications of the ACM, Vol. 15, No. 3, March 1972, pp. 171-176.

4. Hoare, C. A. R. "Monitors: An Operating System Structuring Concept", Communications of the ACM, Vol. 17, No. 10, October 1974, pp. 549-557.